510: Programming Languages

Polymorphism and Universal Types

David Walker

A Critique of Statically Typed Languages

Statically typed languages are sometimes criticized on two grounds:

- Types are obtrusive: types overwhelm the code.

- Types inhibit code re-use: one version for each type.

Enhancing Concision: Type Inference

Implicitly-typed languages allow omission of type information.

```
val axpby =
  fn (a, b) =>
    fn (x, y) => a*x + b*y
```

Officially, the types are present, we just aren't required to specify them.

```
val axpby : int * int -> int * int -> int =
  fn (a:int,b:int):int =>
    fn (x:int,y:int):int => a*x + b*y
```

Enhancing Reuse: Polymorphism

Polymorphism supports **generic** programming:

```
val id : All 'a ('a -> 'a) =
  Fn 'a => fn x:'a => x

val compose :
  All ('a,'b,'c)
    ('b -> 'c) * ('a -> 'b) -> 'a -> 'c =
  Fn ('a,'b,'c) =>
    fn (f:'b->'c, g:'a->'b):'a->'c =>
      fn x:'a => f(g(x))
```

A **type abstraction** is a function that take types as arguments.

Enhancing Reuse: Polymorphism

Polymorphic functions are **instantiated** by applying them to types.

```
val n : int = id[int](3)
val f : int -> string =
  compose[int,char,string](to_string,to_char)
```

Example: Polymorphic Lists

An explicitly-typed, polymorphic `map` function:

```
val map : All ('a,'b)
             ('a -> 'b) -> 'a list -> 'b list =
    Fn ('a, 'b) =>
       fn f : 'a -> 'b =>
          fn l : 'a list =>
             case l
               of Nil['a] => Nil['b]
                | Cons['a](h:'a, t:'a list) =>
                  Cons['b](f h, map['a,'b](f)(t))
```

Polymorphic Type Inference

ML combines implicit typing and polymorphism, allowing us to write:

```
val id = fn x => x

val compose = fn (f,g) => fn x => f(g(x))

fun map f nil = nil
   | f (h::t) => map (f h, map f t)
```

Instantiation is performed automatically:

```
val n = id (3)

val f = compose(to_string,to_char)

val l = map succ [1,2,3]
```

Two Related Concepts

ML does two things for you:

- Infers missing type information in the most general way possible.
- Inserts implicit type abstractions and instantiations.

We'll consider polymorphism as a language mechanism; we will not have time to consider type inference.

Polymorphic MinML

Additions to the abstract syntax:

Types	τ	$::=$	$t \mid \forall t(\tau)$
Expressions	e	$::=$	$\texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end} \mid t\, [\tau]$
Values	v	$::=$	$\texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end}$

The type variable t is bound in the body τ of the polymorphic type $\forall t(\tau)$.

Notation in the literature:

$\Lambda t.e$	*for*	$\texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end}$
$\lambda x{:}\tau.e$	*for*	$\texttt{fn}\, x\texttt{:}\tau\, \texttt{in}\, e\, \texttt{end}$.

Polymorphic Types

Examples:

- $\forall t(t \rightarrow t)$.
- $\forall t(t\,\texttt{list} \rightarrow t\,\texttt{list})$.
- $(\forall t(t \rightarrow t)) \rightarrow (\forall t(t \rightarrow t))$.

Type Formation

Valid type variables and base types:

$$\frac{t \in \Delta}{\Delta \vdash t \ \mathsf{ok}} \qquad \frac{}{\Delta \vdash \mathtt{int}\ \mathsf{ok}} \qquad \frac{}{\Delta \vdash \mathtt{bool}\ \mathsf{ok}}$$

Valid function types:

$$\frac{\Delta \vdash \tau_1 \ \mathsf{ok} \quad \Delta \vdash \tau_2 \ \mathsf{ok}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \ \mathsf{ok}}$$

Context Formation

Valid quantified types:

$$\frac{\Delta \cup \{\, t \,\} \vdash \tau \ \mathsf{ok} \qquad t \notin \Delta}{\Delta \vdash \forall t(\tau) \ \mathsf{ok}}$$

Valid typing contexts:

$$\frac{\Delta \vdash \Gamma(x) \ \mathsf{ok} \qquad (x \in \mathrm{dom}(\Gamma))}{\Delta \vdash \Gamma \ \mathsf{ok}}$$

Static Semantics

Typing judgements: $\Gamma \vdash_{\Delta} e : \tau$.

- Δ is the set of active type variables.
- Γ assigns types to variables.
- e is an expression.
- $\Delta \vdash \Gamma$ ok and $\Delta \vdash \tau$ ok.

Typing Rules

Valid type abstractions:

$$\frac{\Gamma \vdash_{\Delta \cup \{t\}} e : \tau \qquad t \notin \Delta}{\Gamma \vdash_{\Delta} \texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end} : \forall t(\tau)}$$

In words:

- Add t to the active set of type variables, ensuring that it is not already present.
- Type check body with t being active.
- Assign a polymorphic type to the type abstraction.

Typing Rules

Valid type instantiations:

$$\frac{\Delta \vdash \tau \ \mathsf{ok} \qquad \Gamma \vdash_{\Delta} e : \forall t(\tau')}{\Gamma \vdash_{\Delta} e\,[\tau] : \{\tau/t\}\tau'}$$

In words:

- Ensure that the type argument τ is valid.
- Ensure that e is polymorphic.
- Instantiate the type of e by substitution.

Typing Examples

For example,

$$\emptyset \vdash_{\emptyset} \texttt{Fun}\, t\, \texttt{in}\, \texttt{fun}\, f\ \texttt{(}x\texttt{:}t\texttt{)}\texttt{:}t\ =\ x\, \texttt{end} : \forall t(t \to t),$$

because

$$\emptyset \vdash_{\{\,t\,\}} \texttt{fun}\, f\ \texttt{(}x\texttt{:}t\texttt{)}\texttt{:}t\ =\ x : t \to t,$$

because

$$x{:}t \vdash_{\{\,t\,\}} x : t.$$

Typing Examples

For example, $I{:}\forall t(t \to t) \vdash_{\emptyset} I\,\texttt{[int]} : \texttt{int} \to \texttt{int}$, since

- $I{:}\forall t(t \to t) \vdash_{\emptyset} I : \forall t(t \to t)$, and
- $\emptyset \vdash \texttt{int}$ ok, and
- $\{\texttt{int}/t\}t \to t = \texttt{int} \to \texttt{int}$.

An Example

Here's `map` written in PolyMinML extended with lists:

```
Fn t in
  Fn u in
    fun _ (f : t->u) : t list -> u list is
      fun loop (l : t list) : u list is
        listcase l of
          nil => nil
        | cons(x,l) => cons (f x, loop l)
```

Dynamic Semantics

Main ideas:

- Type abstractions are values (just like ordinary abstractions).

- Type instantiation is an instruction step.

We'll use SOS to specify the dynamic semantics of polymorphic MinML.

Dynamic Semantics

New instruction:

$$\frac{}{\texttt{Fun}\, t\, \texttt{in}\, e\, \texttt{end}\, \texttt{[}\tau\texttt{]} \mapsto \{\tau/t\}e}$$

New search rule:

$$\frac{e \mapsto e'}{e\,\texttt{[}\tau\texttt{]} \mapsto e'\,\texttt{[}\tau\texttt{]}}$$

Type Safety

Theorem 1

If $e : \tau$, then either e is a value or there exists $e' : \tau$ such that $e \mapsto e'$.

Proof sketch:

- Prove preservation by induction on evaluation.

- Prove progress by induction on typing.

- A closed value of type $\forall t(\tau)$ is a type abstraction: $\mathtt{Fun}\, t \,\mathtt{in}\, e \,\mathtt{end}$.

“Shallow” Polymorphism

ML provides only **prenex**, or **shallow**, polymorphism: $\forall t_1,\ldots,t_n(\tau)$, where τ is not a quantified type.

- A **polytype** σ is either a **monotype** τ, or a quantified polytype $\forall t(\sigma)$.

- A **monotype** τ is a MinML type, possibly involving type variables (*e.g.*, $t{\rightarrow}t$).

This limitation is necessary to support “full” type inference (no types are ever required.)

“Deep” Polymorphism

PolyMinML admits **deep** polymorphism:

- Can pass polymorphic functions as arguments and return them as results.

- Can build lists (or other aggregates) of polymorphic functions.

- Can store polymorphic functions in reference cells.

Polymorphic References

Consider the type $T = \forall t((t \to t)\,\texttt{ref})$.

- Elements are polymorphic functions f.

- Instantiation of f to τ yields a value of type $\tau \to \tau\,\texttt{ref}$.

Type of **polymorphic functions yielding reference cells**.

Polymorphic References

Example function f of this type:

```
Fn t in ref (fun _(x:t):t in x)
```

This is a **value**.

Each time we instantiate f we get a **new** reference cell.

- f `[int]` : $($`int`$\rightarrow$`int`$)$ `ref`.
- f `[bool]` : $($`bool`$\rightarrow$`bool`$)$ `ref`.

Polymorphic References

Now consider the type $U = (\forall t(t \to t))\,\mathtt{ref}$.

- Elements are reference cells holding a polymorphic function.

- This is not a prenex polymorphic type.

Type of **reference cells containing polymorphic functions**.

Polymorphic References

Example value x of this type:

- $\texttt{ref}(\texttt{Fun}\,t\,\texttt{in}\,\texttt{fun}\,\_\,\texttt{(}x\texttt{:}t\texttt{):}t\ =\ x\,\texttt{end})$.

- A **single** reference cell containing the polymorphic identity.

Polymorphic References

Consider the following example:

```
let
    val r : ('a -> 'a) ref = ref (fn x => x)
in
    r := succ ; (!r)(true)
end
```

In early versions of ML this was type correct, yet it's clearly not safe!

Polymorphic References

With explicit polymorphism the problem is revealed.

What is `r`?

- A **polymorphic function yielding a reference cell**? Or . . .

- A **reference cell containing a polymorphic function**?

Polymorphic References

Version 1: a polymorphic function yielding reference cells.

```
let val r = Fn t in ref (fn x:t => x)
in  r[int] := succ ; (! (r[bool]))(true) end
```

Each instance of `r` yields a **fresh** reference cell.

- The set and the get are performed on distinct cells.
- Sound, but not particularly useful.

Polymorphic References

Version 2: a reference cell containing a polymorphic function.

```
let val r = ref (Fn t in fn x:t => x)
in r := succ ; (! r)[bool](true) end
```

This is **type incorrect**.

- `succ` has type `int`$\rightarrow$`int`.

- Assignment requires value of type $\forall t(t \rightarrow t)$.

Unsound, and duly rejected by the type checker.

Representing Data Structures

A rich variety of types are **representable** in PolyMinML.

- Product (tuple) types.
- Sum (disjoint union) types.
- Natural numbers.
- Lists, streams, trees.

Representing Data Structures

To be representable means that

- We can **define** the type in terms of PolyMinML types.
- We can **define** the introduction and elimination forms for the types.

Key idea: **active**, rather than **passive**, data.

- Data structures respond to **messages**.
- Elimination forms send messages to the data.

Representing Products

Type:

$$\tau_1 \texttt{*} \tau_2 := \forall t((\tau_1 \to \tau_2 \to t) \to t)$$

Pairs:

$$\texttt{(}v_1\texttt{,}v_2\texttt{)} := \Lambda\, t.\lambda h{:}\tau_1 \to \tau_2 \to t.h\texttt{(}v_1\texttt{)}\texttt{(}v_2\texttt{)}$$

Split:

$$\texttt{split}\, e\, \texttt{as}\, \texttt{(}x\texttt{,}y\texttt{)}\, \texttt{in}\, e'\, \texttt{end} := e\, \texttt{[}\tau'\texttt{]}\lambda x{:}\tau_1.\lambda y{:}\tau_2.e'$$

Representing Products

Idea: the pair take a **result type** and a **handler** as arguments, and passes the components of the pair to the handler to compute the result.

Check:

$$\texttt{split}\,(v_1\texttt{,}v_2)\,\texttt{as}\,(x\texttt{,}y)\,\texttt{in}\,e'\,\texttt{end}$$

$$= (v_1\texttt{,}v_2)\,[\tau']\,(\lambda x{:}\tau_1.\lambda y{:}\tau_2.e')$$

$$\mapsto (\lambda x{:}\tau_1.\lambda y{:}\tau_2.e')(v_1)(v_2)$$

$$\mapsto \{v_1, v_2/x, y\}e'$$

This is the correct behavior!

Representing Sums

Type:

$$\tau_1\texttt{+}\tau_2 := \forall t((\tau_1 \to t) \to (\tau_2 \to t) \to t)$$

Injections:

$$\begin{array}{lcl}
\texttt{inl}_{\tau_1\texttt{+}\tau_2}(v) & := & \Lambda\, t.\lambda f{:}\tau_1 \to t.\lambda g{:}\tau_2 \to t.f\,(v) \\
\texttt{inr}_{\tau_1\texttt{+}\tau_2}(v) & := & \Lambda\, t.\lambda f{:}\tau_1 \to t.\lambda g{:}\tau_2 \to t.g\,(v)
\end{array}$$

Case analysis:

$$\begin{array}{c}
\texttt{case}_\tau\, e\, \texttt{of}\, \texttt{inl}\,(x\,{:}\,\tau_1)\, \texttt{=>}\, e_1 \mid \texttt{inr}\,(y\,{:}\,\tau_2)\, \texttt{=>}\, e_2\, \texttt{end} \\
:= \\
e\, [\tau]\, (\lambda x{:}\tau_1.e_1)\, (\lambda x{:}\tau_2.e_2)
\end{array}$$

Representing Natural Numbers

Type:

$$\mathtt{nat} := \forall t((t \rightarrow t) \rightarrow (t \rightarrow t))$$

Numbers:

$$\begin{array}{rcl} \mathtt{zero} & := & \Lambda\, t.\lambda s{:}t \rightarrow t.\lambda b{:}t.b \\ \mathtt{succ}(n) & := & \Lambda\, t.\lambda s{:}t \rightarrow t.\lambda b{:}t.s\mathtt{(}n\ \mathtt{[}t\mathtt{]}\ \mathtt{(}s\mathtt{)}\ \mathtt{(}b\mathtt{))} \end{array}$$

Rerpresenting Natural Numbers

Some arithmetic:

$$
\begin{aligned}
m + n &:= n\,[\mathtt{nat}](\lambda p{:}\mathtt{nat}.\mathtt{succ}(p))(m) \\
m * n &:= n\,[\mathtt{nat}](\lambda p{:}\mathtt{nat}.p + m)(\mathtt{zero}) \\
m^n &:= n\,[\mathtt{nat}](\lambda p{:}\mathtt{nat}.p * m)(\mathtt{succ}(\mathtt{zero}))
\end{aligned}
$$

Summary

Polymorphism supports generic programming.

Universal types formalize polymorphism.

Polymorphism may be used to encode data structures.